MICROSOFT

BASIC-80

8K CASSETTE & ROM VERSIONS

FOR

NETRONICS EXPLORER COMPUTER

reference manual

NETRONICS RESEARCH AND DEVELOPMENT LIMITED
333 LITCHFIELD ROAD (RTE. 202) NEW MILFORD, CONNECTICUT 06776
(203) 354-9375

BASIC-80 REFERENCE MANUAL

CONTENTS

INTRODUCTION

READ BEFORE PROCEEDING

The manual is divided into three large chapters plus a number of appendices. Chapter 1 covers a variety of topics, largely pertaining to information representation when using BASIC-80. Chapter 2 contains the syntax and semantics of every command and statement in BASIC-80, ordered alphabetically. Chapter 3 describes all of BASIC-80's intrinsic functions, also ordered alphabetically. The appendices contain lists of error messages ASCII codes, math functions, and helpful information on assembly language subroutines.

Information contained in the manual occasionally makes reference to a Extended and Disk version which are larger versions of Microsoft Basic. This information is supplied so that when you encounter programs written for the larger version you will understand how to modify them to run in your 8K version.

The Netronics ROM version is located at CØØØH to DFFFH and requires memory from ØØØØH up. The cassette version is automatically loaded at ØØØØH to 1FFFH and requires memory from 2ØØØH up. The label used to load the cassette is BA (monitor command .LBA).

To HALT a program execution or the listing function a hardware modification is required on your Explorer. Simply connect a jumper from J2 pin 22 to U105 pin 28. The program tests this input port at the beginning of each line in both the RUN and LIST mode. Depressing the Interrupt button will stop program execution and listing function. Depressing any keyboard switch will continue operation while depressing a control C will branch the Basic to the command mode.

CHAPTER 1

GENERAL INFORMATION ABOUT BASIC-80

INITIALIZATION

The Explorer monitor is used to load (cassette version) and to initialize the Basic program. Set the stack pointer to F88Ø and the program counter to ØØØØ then execute the G command. The system will respond (Set PC to C ØØØ in Rom version)

memory size: You enter the address of the highest free memory you wish to use for programs (CR) (Defalts to highest available RAM if no limit set)

terminal width: You enter the number of characters per line that your terminal generates (CR) (Defalts to 72 characters if no limit set)

The program will then issue the Microsoft Sign-on plus indicate the amount of free memory available for Basic programs followed by (OK) which indicates that BASIC-80 is in the command mode.

MODES OF OPERATION

When BASIC-80 is initialized, it types the prompt "OK". "OK" means BASIC-80 is at command level, that is, it is ready to accept commands. At this point, BASIC-80 may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC commands and statements are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

nnnnn BASIC statement [:BASIC statement...] carriage return

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of:

72 characters in 8K BASIC-80

LINE NUMBERS

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529.

CHARACTER SET

The BASIC-80 character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in BASIC-80 are the upper case and lower case letters of the alphabet.

The numeric characters in BASIC-80 are the digits 0 through 9.

The following special characters and terminal keys are recognized by BASIC-80:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number or pound sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark
:	Colon
&	Ampersand
?	Question mark
	Less than
	Greater than
rubout	Deletes last character typed
carriage return	Terminates input of a line

GENERAL INFORMATION ABOUT BASIC-80

CONTROL CHARACTERS

The following control characters are in BASIC-80:

Control C	Interrupts program execution and returns to BASIC-80 command level. (After Explorer Interrupt RST. 7.5 button is depressed).
Control-G	Rings the bell at the terminal.
Control-H	Backspace. Deletes the last character typed.
Control-R	Retypes the line that is currently being typed.
Control-U	Deletes the line that is currently being typed.
Explorer Interrupt	Suspends program execution. Any key resumes program execution after a Explorer Interrupt.

CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants	Whole numbers between -32768 and +32767. Integer constants do not have decimal points.
2. Fixed Point Constants	Positive or negative real numbers, i.e., numbers that contain decimal points.
3. Floating Point constants	Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The exponent must be in the range -38 to +38. Examples: 235.988E-7 = .0000235988 2359E6 = 2359000000

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E

VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

VARIABLE NAMES AND DECLARATION CHARACTERS

BASIC-80 variable names may be any length, however, in the 8K version, only the first two characters are significant. In the Extended and Disk versions, up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point is allowed in Extended and Disk variable names. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word. The Extended and Disk versions allow embedded reserved words; the 8K version does not. If a variable begins with Fn, it is assumed to be a call to a user-defined function. Reserved words include all BASIC-80 commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$= "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

N$	declares a string value
ABC	represents a single precision value

ARRAY VARIABLES

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on.

EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC-80 may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
∧	Exponentiation	X∧Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
X+2Y	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X∧2)∧Y
X^{Y^Z}	X∧(Y∧Z)
X(-Y)	X*(-Y) Two consecutive operators must be separated by parentheses.

Overflow And Division By Zero -

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then used to make a decision regarding program flow. (See IF).

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z.

```
IF SIN(X)<0 GOTO 1000
```

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF).

example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator maybe used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC-80 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC-80's intrinsic functions are described in Chapter 3.

BASIC-80 also allows "user defined" functions that are written by the programmer. See DEF FN

String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=     <>     <     >     <=     >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"      where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key or with Control-H. Rubout surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC-80 will automatically replace the old line with the new line.

To delete the entire program that is currently residing in memory, enter the NEW command. NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If BASIC-80 detects an error that causes program execution to terminate, an error message is printed. In the 8K version, only the error code is printed. In the Extended and Disk versions, the entire error message is printed. For a complete list of BASIC-80 error codes and error messages, see Appendix .

CHAPTER 2

BASIC-80 COMMANDS AND STATEMENTS

All of the BASIC-80 commands and statements are described in this chapter. Each description is formatted as follows:

Format: Shows the correct format for the instruction. See below for format notation.

Versions: Lists the versions of BASIC-80 in which the instruction is available.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

Format Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.
2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.
4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

CLEAR

Format: CLEAR [expression]

Versions: 8K

Purpose: To set all numeric variables to zero and all string variables to null, and optionally, to set the amount of string space.

CLOAD

Formats: CLOAD <filename>

CLOAD? <filename>

CLOAD* <array name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To load a program or an array from cassette tape into memory.

Remarks: CLOAD executes a NEW command before it loads the program from cassette tape. <filename> is the string expression or the first character of the string expression that was specified when the program was CSAVEd.

CLOAD? verifies tapes by comparing the program currently in memory with the file on tape that has the same filename. If they are the same, BASIC-80 prints Ok. If not, BASIC-80 prints NO GOOD.

CLOAD* loads a numeric array that has been saved on tape. The data on tape is loaded into the array called <array name> specified when the array was CSAVE*ed.

CLOAD and CLOAD? are always entered at command level as direct mode commands. CLOAD* may be entered at command level or used as a program statement. Make sure the array has been DIMensioned before it is loaded. BASIC-80 always returns to command level after a CLOAD, CLOAD? or CLOAD* is executed. Before a CLOAD is executed, make sure the cassette recorder is properly connected and in the Play mode, and the tape is possitioned correctly.

See also CSAVE,

NOTE: CLOAD and CSAVE are not included in all implementations of BASIC-80.

Example: CLOAD "MAX2"

Loads file "M" into memory.

CONT

Format: CONT

Versions: 8K, Extended, Disk

Purpose: To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. With the Extended and Disk versions, CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break. In 8K BASIC-80, execution cannot be CONTinued if a direct mode error has occurred during the break.

CSAVE

Formats: CSAVE <string expression>

CSAVE* <array variable name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To save the program or an array currently in memory on cassette tape.

Remarks: Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, BASIC-80 saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE* <array variable name> is executed, BASIC-80 saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest.

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the Record mode.

See also CLOAD.

NOTE: After the program has been loaded onto the cassette the terminal will display 2 or 3 random characters and return the Basic program to the command mode.

Example: CSAVE "TIMER"

Saves the program currently in memory on cassette under filename "T".

DATA

Format: DATA <list of constants>

Versions: 8K, Extended, Disk

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s).

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement

Example: See examples in Section READ.

DEF FN

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Versions: 8K, Extended, Disk

Purpose: To define and name a function that is written by theuser.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call. (Remember, in the 8K version only one argument is allowed in a function call, therefore the DEF FN statement will contain only one variable.)

In Extended and Disk BASIC-80, user-defined functions may be numeric or string; in 8K, user-defined string functions are not allowed. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
      .
      .
410 DEF FNAB (X,Y)=X 3/Y^2
420 T=FNAB(I,J)
      .
      .
```

DIM

Format: DIM <list of subscripted variables>

Versions: 8K, Extended, Disk

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example: 10 DIM A(20)

END

Format: END

Versions: 8K, Extended, Disk

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC-80 always returns to command level after an END is executed.

FOR...NEXT

Format: FOR <variable>=x TO y [STEP z]
.
.
.
NEXT [<variable>][,<variable>...]

where x, y and z are numeric expressions.

Versions: 8K, Extended, Disk

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: When the FOR statement is encountered for the first time, the expressions are evaluated. The variable is set to the value of X which is called the initial value. BASIC then executes the statements which follow the FOR statement in the usual manner. When a NEXT statement is encountered, the step Z is added to the variable which is the tested against the final value Y. If Z, the step, is positive and the variable is less than or equal to the final value or if the step is negative and the variable is greater than or equal to the final value, then BASIC branches back to the statement immediately following the FOR statement. Otherwise, execution proceeds with the statement following the NEXT. If the step is not specified, it is assumed to be 1. Examples:

```
10 FOR I=2 TO 11      The loop is executed 10
   times with the variable I taking on each
   integral value from 2 to 11.
20 FOR V=1 TO 9.3     This loop will execute 9
   times until V is greater than 9.3
30 FOR V=10*N TO 3.4/Z STEP SQR (R) the initial,
   final and step expressions need not be inte-
   gral, but they will be evaluated only once,
   before looping begins.
40 FOR V=9 TO 1 STEP -1 This loop will be exe-
   cuted 9 times.
```

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR... NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable (s) in the NEXT statement may be

omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

GOSUB...RETURN

Format:

```
GOSUB <line number>
    .
    .
    .
RETURN
```

Versions: 8K, Extended, Disk

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC-80 to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

GOTO

Format: GOTO <line number>

Versions: 8K, Extended, Disk

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 4           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
?Out of data in 10
Ok
```

IF...THEN[...ELSE] AND IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>

[ELSE <statement(s)> | <line number>]

Format: IF <expression> GOTO <line number>

[ELSE <statement(s)> | <line number>]

Versions: 8K, Extended, Disk

NOTE: The ELSE clause is allowed only in Extended and Disk versions.

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. (ELSE is allowed only in Extended and Disk versions.) Extended and Disk versions allow a comma before THEN.

Nesting of IF Statements

In the Extended and Disk versions, IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
     THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

INPUT

Format: INPUT [<"prompt string">;]<list of variables>

Versions: 8K, Extended, Disk

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the messsage "?Redo from start" to be printed.

In the 8K version, INPUT is illegal in the direct mode.

Examples:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?  5      (The 5 was typed in by the user
           in response to the question mark.)
 5 SQUARED IS 25
Ok


LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS?  7.4  (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

LET

Format: [LET] <variable>=<expression>

Versions: 8K, Extended, Disk

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
     .
     .
     .
```

or

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
     .
     .
     .
```

LIST

Format: LIST [<line number>]

Versions: 8K, Extended, Disk

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC-80 always returns to command level after a LIST is executed.

Format: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by using Explorer interrupt followed by a Control-C)

NEW

Format: NEW

Versions: 8K, Extended, Disk

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC-80 always returns to command level after a NEW is executed.

NULL

Format: NULL <integer expression>

Versions: 8K, Extended, Disk

Purpose: To set the number of nulls to be printed at the end of each line.

Remarks: For 10-character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletypes and Teletype-compatible CRTs. <integer expression> should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
     .
     .
     .
```

Two null characters will be printed after each line.

ON...GOSUB AND ON...GOTO

Format: ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Versions: 8K, Extended, Disk

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

Example: 100 ON L-1 GOTO 150,300,320,390

OUT

Format: OUT I,J
where I and J are integer expressions in the range 0 to 255.

Versions: 8K, Extended, Disk

Purpose: To send a byte to a machine output port.

Remarks: The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example: 100 OUT 32,100

POKE

Format: POKE I,J
where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. In the 8K version, I must be less than 32768. In the Extended and Disk versions, I must be in the range 0 to 65536.

With the 8K version, data may be POKEd into memory locations above 32768 by supplying a negative number for I. The value of I is computed by subtracting 65536 from the desired address. For example, to POKE data into location 45000, I = 45000-65536, or -20536.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.27.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

PRINT

Format: PRINT [<list of expressions>]

Versions: 8K, Extended, Disk

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC-80 divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC-80 goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:
```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10            0            -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:
```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:
```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

READ

Format: READ <list of variables>

Versions: 8K, Extended, Disk

Purpose: To read values from a DATA statement and assign them to variables.

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statment. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE)

Example 1:

```
 .
 .
 .
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
 .
 .
 .
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE          ZIP
DENVER,       COLORADO       80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

REM

Format: REM <remark>

Versions: 8K, Extended, Disk

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

In the Extended and Disk versions, remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Example:

```
    .
    .
    .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
    .
    .
    .
```

or, with Extended and Disk versions:

```
    .
    .
    .
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
    .
    .
    .
```

RESTORE

Format: RESTORE [<line number>]

Versions: 8K, Extended, Disk

Purpose: To allow DATA statements to be reread from a specified point.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:
```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
    .
    .
    .
```

RUN

Format 1: RUN [<line number>]

Versions: 8K Extended, Disk

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC-80 always returns to command level after a RUN is executed.

Example: RUN

STOP

Format: STOP

Versions: 8K, Extended, Disk

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

```
Break in line nnnnn
```

Unlike the END statement, the STOP statement does not close files.

BASIC-80 always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

Example:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

WAIT

Format: WAIT <port number>, I[,J]
where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC-80 loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example:
```
100 WAIT 32,2
```

CHAPTER 3

BASIC-80 FUNCTIONS

The intrinsic functions provided by BASIC-80 are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC-80 will round the fractional portion and use the resulting integer.

ABS

Format: ABS(X)

Versions: 8K, Extended, Disk

Action: Returns the absolute value of the expression X.

Example:
```
PRINT ABS(7*(-5))
 35
Ok
```

ASC

Format: ASC(X$)

Versions: 8K, Extended, Disk

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example:
```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

ATN

Format: ATN(X)

Versions: 8K, Extended, Disk

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:
```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

CHR$

Format: CHR$(I)

Versions: 8K, Extended, Disk

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix L.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a CRT screen and return the cursor to the home position.

Example:
```
PRINT CHR$(66)
B
Ok
```
See the ASC function for ASCII-to-numeric conversion.

COS

Format: COS(X)

Versions: 8K, Extended, Disk

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:
```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

EXP

Format: EXP(X)

Versions: 8K, Extended, Disk

Action: Returns e to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

FRE

Format: FRE(0)
FRE(X$)

Versions: 8K, Extended, Disk

Action: Arguments to FRE are dummy arguments. If the argument is 0 (numeric), FRE returns the number of bytes in memory not being used by BASIC-80. If the argument is a string, FRE returns the number of free bytes in string space.

Example:
```
PRINT FRE(0)
 14542
Ok
```

INP

Format: INP(I)

Versions: 8K, Extended, Disk

Action: Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement

Example:
```
100 A=INP(255)
```

INT

Format: INT(X)

Versions: 8K, Extended, Disk

Action: Returns the largest integer <=X.

Examples:
```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

LEFT$

Format: LEFT$(X$,I)

Versions: 8K, Extended, Disk

Action: Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example:
```
10 A$ = "BASIC-80"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions.

LEN

Format: LEN(X$)

Versions: 8K, Extended, Disk

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example:
```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

LOG

Format: LOG(X)

Versions: 8K, Extended, Disk

Action: Returns the natural logarithm of X. X must be greater than zero.

Example:
```
PRINT LOG(45/7)
 1.86075
Ok
```

MID$

Format: MID$(X$,I[,J])

Versions: 8K, Extended, Disk

Action: Returns a string of length J characters from X$ beginning with the Ith character. I and J must be in the range 0 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

Example:
```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions.

PEEK

Format: PEEK(I)

Versions: 8K, Extended, Disk

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. With the 8K version of BASIC-80, I must be less than 32768. To PEEK at a memory location above 32768, subtract 65536 from the desired address. With Extended and Disk BASIC-80, I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement.

Example:
```
A=PEEK(32765)
```

3.28 POS

Format: POS(I)

Versions: 8K, Extended, Disk

Action: Returns the current cursor position. The leftmost position is 0. X is a dummy argument.

Example:
```
IF POS(X)>60 THEN PRINT CHR$(13)
```

RIGHT$

Format: RIGHT$(X$,I)

Versions: 8K, Extended, Disk

Action: Returns the rightmost I characters of string X$. If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Example:
```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$(A$,8)
RUN
BASIC-80
Ok
```

Also see the MID$ and LEFT$ functions.

RND

Format: RND[(X)]

Versions: 8K, Extended, Disk

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Section 2.53). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:
```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

SGN

Format: SGN(X)

Versions: 8K, Extended, Disk

Action: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

SIN

Format: SIN(X)

Versions: 8K, Extended, Disk

Action: Returns the sine of X in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

Example:
```
PRINT SIN(1.5)
 .997495
Ok
```

SPC

Format: SPC(I)

Versions: 8K, Extended, Disk

Action: Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255.

Example:
```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

SQR

Format: SQR(X)

Versions: 8K, Extended, Disk

Action: Returns the square root of X. X must be >=0.

Example:
```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

STR$

Format: STR$(X)

Versions: 8K, Extended, Disk

Action: Returns a string representation of the value of X.

Example:
```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

TAB

Format: TAB(I)

Versions: 8K, Extended, Disk

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                      AMOUNT

G. T. JONES               $25.00
Ok
```

TAN

Format: TAN(X)

Versions: 8K, Extended, Disk

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
10 Y = Q*TAN(X)/2
```

USR

Format: USR[<digit>](X)

Versions: 8K, Extended, Disk

Action: Calls the user's assembly language subroutine with the argument X. <digit> is allowed in the Extended and Disk versions only. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix

Example:
```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
   .
   .
   .
```

APPENDIX A

New Features in BASIC-80

The execution of BASIC programs written under Microsoft BASIC, release 4.51 and earlier may be affected by some of the new features in this release. Before attempting to run such programs, check for the following:

1. Conversion from floating point to integer values results in rounding, as opposed to truncation.

2. The body of a FOR...NEXT loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

3. Division by zero and overflow no longer produce fatal errors.

4. The RND function has been changed so the RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN.

4. The rules for PRINTing single precision numbers have been changed. (See Section PRINT).

5. If the argument to ON...GOTO is out of range, an error message results and execution halts.

6. The at-sign and underscore are no longer used as editing characters.

APPENDIX B

Assembly Language Subroutines

All versions of BASIC-80 have provisions for interfacing with assembly language subroutines. The USR Function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

NOTE

The addresses of Assembly language routines are stored at the following locations:

Routines	Cassette Version	Rom Version
DEINT	Ø944	C944
GIVABF	1ØE4	D0E4
USRLOC	2ØØ4	ØØØ4

MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). BASIC uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the system monitor, or the BASIC POKE statement, or (if the user has the MACRO-80 or FORTRAN-80 package) routines may be assembled with MACRO-80 and loaded using LINK-80.

USR FUNCTION CALLS - 8K BASIC

The starting address of the assembly language subroutine must be stored in USRLOC, a two-byte location in memory that is supplied individually with different implementations of BASIC-80. With 8K BASIC, the starting address may be POKEd into USRLOC. Store the low order byte first, followed by the high order byte.

The function USR will call the routine whose address is in USRLOC. Initially USRLOC contains the address of ILLFUN, the routine that gives the "Illegal function call" error. Therefore, if USR is called without changing the address in USRLOC, an "Illegal function call" error results.

The format of a USR function call is

```
USR(argument)
```

where the argument is a numeric expression. To obtain the argument, the assembly language subroutine must call the routine DEINT. DEINT places the argument into the D,E register pair as a 2-byte, 2's complement integer. (If the argument is not in the range -32768 to 32767, an "Illegal function call" error occurs.)

To pass the result back from an assembly language subroutine, load the value in register pair [A,B], and call the routine GIVABF. If GIVABF is not called, USR(X) returns X. To return to BASIC, the assembly language subroutine must execute a RET instruction.

For example, here is an assembly language subroutine that multiplies the argument by 2:

```
USRSUB: CALL DEINT      ;put arg in D,E
        XCHG            ;move arg to H,L
        DAD H           ;H,L=H,L+H,L
        MOV A,H         ;move result to A,B
        MOV B,L
        JMP GIVABF      ;pass result back and RETurn
```

Note that valid results will be obtained from this routine for arguments in the range -16384<=x<=16383. The single instruction JMP GIVABF has the same effect as:

```
CALL GIVABF
RET
```

To return additional values to the program, load them into memory and read them with the PEEK function.

There are several methods by which a program may call more than one USR routine. For example, the starting address of each routine may be POKEd into USRLOC prior to each USR call, or the argument to USR could be an index into a table of USR routines.

INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register A-L and the PSW. Interrupts should always be re-enabled before returning from the subroutine, since an interrupt automatically disables all further interrupts once it is received. The user should be aware of which interrupt vectors are free in the particular version of BASIC that has been supplied. Note to CP/M users: in CP/M BASIC, all interrupt vectors are free.)

APPENDIX C

Converting Programs to BASIC-80

If you have programs written in a BASIC other than BASIC-80, some minor adjustments may be necessary before running them with BASIC-80. Here are some specific things to look for when converting BASIC programs.

STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the BASIC-80 statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for BASIC-80 string concatenation.

In BASIC-80, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	BASIC-80
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	8K BASIC-80
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1);X$;MID$(A$,J+1)

	Ext. and Disk BASIC-80
A$(I)=X$	MID$(A$,1,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. BASIC-80 would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

MULTIPLE STATEMENTS

Some BASICs use a backslash ( ) to separate multiple statements on a line. With BASIC-80, be sure all statements on a line are separated by a colon (:).

MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

APPENDIX D

Summary of Error Codes and Error Messages

Code	Number	Message
BS	9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
CN	17	Can't continue An attempt is made to continue a program that: 1. has halted due to an error, 2. has been modified during a break in execution, or 3. does not exist.
DD	10	Redimensioned array Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. a negative or unreasonably large subscript 2. a negative or zero argument with LOG 3. a negative argument to SQR 4. a negative mantissa with a non-integer exponent

5. a call to a USR function for which the starting address has not yet been given

6. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

ID 12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

NF 1 NEXT without FOR
A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.

OD 4 Out of data
A READ statement is executed when there are no DATA statements with unread data remaining in the program.

OM 7 Out of memory
A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

OS 14 Out of string space
String variables exceed the allocated amount of string space. Use CLEAR to allocate more string space, or decrease the size and number of strings.

OV 6 Overflow
The result of a calculation is too large to be represented in BASIC-80's number format. If underflow occurs, the result is zero and execution continues without an error.

SN 2 Syntax error
A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).

ST 16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

TM 13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

RG	3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
UF	18	Undefined user function A USR function is called before the function definition (DEF statement) is given.
UL	8	Undefined line A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.
/0	11	Division by zero A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

APPENDIX E

Mathematical Functions

Derived Functions

Functions that are not intrinsic to BASIC-80 may be calculated as follows:

Function	BASIC-80 Equivalent
SECANT	`SEC(X)=1/COS(X)`
COSECANT	`CSC(X)=1/SIN(X)`
COTANGENT	`COT(X)=1/TAN(X)`
INVERSE SINE	`ARCSIN(X)=ATN(X/SQR(-X*X+1))`
INVERSE COSINE	`ARCCOS(X)=-ATN (X/SQR(-X*X+1))+1.5708`
INVERSE SECANT	`ARCSEC(X)=ATN(X/SQR(X*X-1))` `+SGN(SGN(X)-1)*1.5708`
INVERSE COSECANT	`ARCCSC(X)=ATN(X/SQR(X*X-1))` `+(SGN(X)-1)*1.5708`
INVERSE COTANGENT	`ARCCOT(X)=ATN(X)+1.5708`
HYPERBOLIC SINE	`SINH(X)=(EXP(X)-EXP(-X))/2`
HYPERBOLIC COSINE	`COSH(X)=(EXP(X)+EXP(-X))/2`
HYPERBOLIC TANGENT	`TANH(X)=EXP(-X)/EXP(X)+EXP(-X))*2+1`
HYPERBOLIC SECANT	`SECH(X)=2/(EXP(X)+EXP(-X))`
HYPERBOLIC COSECANT	`CSCH(X)=2/(EXP(X)-EXP(-X))`
HYPERBOLIC COTANGENT	`COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1`
INVERSE HYPERBOLIC SINE	`ARCSINH(X)=LOG(X+SQR(X*X+1))`
INVERSE HYPERBOLIC COSINE	`ARCCOSH(X)=LOG(X+SQR(X*X-1)`
INVERSE HYPERBOLIC TANGENT	`ARCTANH(X)=LOG((1+X)/(1-X))/2`
INVERSE HYPERBOLIC SECANT	`ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)`
INVERSE HYPERBOLIC COSECANT	`ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X`
INVERSE HYPERBOLIC COTANGENT	`ARCCOTH(X)=LOG((X+1)/(X-1))/2`

APPENDIX F

ASCII Character Codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	<
010	LF	053	5	096	'
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout